Ministerul Educatiei, Culturii şi Cercetarii al Republicii Moldova Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Ingineria Software şi Automatica

Report

Laboratory work Nr.2

Course: Operating Systems

Implemented by:

Alexandra Konjevic, FAF-213

Verified by:

Rostislav Călin

Chișinău – 2023

Subject: Writing text using assembly

Objectives:

Create a program in assembler which will "echo" what is typed from the keyboard. Each ASCII character which will be pressed from the keyboard should appear on the screen and the cursor should move to the next position. Special actions need to be implemented only for 2 special keys from the keyboard:

1. "backspace key" - in this case symbol from the left side of the cursor should disappear and the cursor should be moved one position back (If the cursor already is in the first position, then nothing should happen. Special case is if the cursor is on the next line, then when is pressed Backspace in the first column, then cursor should move to the previous line in last column);

2. "enter key" - in this case all previously introduced string should be printed to the screen starting with the next line and after one "empty" line (but if "enter key" will be pressed as the first key, in this case NO "empty" line should be added and the action should just go to the next line)

(OPTIONAL)

The maximum length of input string should not exceed 256 characters. If the user will want to input more than 256 characters than the input should be stopped and, in this case, only "backspace" or "enter" keys should be accepted.

Compiled program should be used in order to create a floppy image and it should be bootable. Use this image to boot the OS in a VirtualBox VM and the text should appear on the screen.

Implementation:

First of all, I defined two constants in the program: backspace key and enter key, by passing the hexadecimal values `*0x08*` and `*0x0D*` to the constants `*BACKSPACE*` and `*ENTER*`:

```
%define BACKSPACE 0x08
%define ENTER 0x0D
```

After that, I wrote the `*start*` section of the program:

```
start:
    mov ah, 0; set the video mode
    mov al, 3; 80x25 text mode
    int 10h ; Video BIOS interrupt to set the mode

    mov cx, 0; character counter
    mov bx, buffer; buffer pointer
```

First of all, I set the video mode, using the 00H function of the interruption 10h. It initializes the video hardware to display in the specified video mode, and expects the 00H function is the ah register, and the optional value in al - the video mode number (which I set to 3, the 80x25 text mode). The 80x25 text mode is a common text mode used in older PCs. It sets the screen to display 80 columns and 25 rows of text characters. Each character cell on the screen can display a single character from the ASCII character set. After that, I initialized two registers for later use: the cx register stores the count of the characters introduced in input, and the next line loads the memory address of the *`buffer`* label in the bx register.

The labels that I have in this program, are:

```
cursor_coords:
cursor_x db 0
cursor_y db 0

row db 0

buffer:
resb 256
```

- *`cursor_coords`*: This line defines a label named *`cursor_coords`*. Labels are used to mark specific memory locations or code sections in assembly language.
- *`cursor_x db 0`* and *`cursor_y db 0`*: These lines allocate memory for storing the x and y coordinates of a cursor on the screen. *`db`* stands for "define byte". It reserves a single byte of memory for each variable (*`cursor_x`* and *`cursor_y`*). Both *`cursor_x`* and *`cursor_y`* is initialized to 0, indicating that the cursor is positioned at the coordinates (0, 0) on the screen.
- *`row db 0`*: This line allocates a single byte of memory for storing a variable named row, which keeps track of a row number of the cursor.
- *`buffer: resb 256`*: This section allocates a block of memory labeled *`buffer`* that reserves space for storing a sequence of characters. *`resb`* stands for "reserve bytes". It allocates a block of memory of the specified size, in this case, 256 bytes (*`resb 256`*), for the buffer.

1. **`read_char`** function:

```
read_char:
    mov ah, 0 ; Reset AH for keyboard interrupt
    int 16h ; BIOS interrupt to read a character from the keyboard

    cmp al, BACKSPACE; if the character is backspace
    je .call_handle_backspace; jump to handle_backspace
    cmp al, ENTER; if the character is enter
    je .call_handle_enter; jump to handle_enter
    jmp .call_handle_symbol; jump to handle_symbol


    .call_handle_backspace:
    call handle_backspace; handle the character
    jmp read_char; read another character

    .call_handle_enter:
    call handle_enter; handle the enter character
    jmp read_char; read another character

    .call_handle_symbol:
    call handle_symbol; handle the character
    jmp read_char; read another character
```

The function begins with the instruction 00h of the interruption 16h. This interruption is used for handling input from the keyboard. Keystrokes are processed asynchronously (in the background). As each keystroke is received from the keyboard, it is processed by int 09h and placed into a circular queue in the BIOS Data Area. The 00h function reads and waits for the next keystroke. If a keystroke is available in the keyboard buffer, the function removes it from the queue and returns it in AH and AL. If no key is available, it waits for a keystroke.

So, the value from AL is compared next to the values of the backspace key, and if they are equal, the function jumps to the *`.call_handle_backspace`*, which calls the *`handle_backspace`* function, and after that, call itself (the *`read_char`* function). The same is done for the enter key.

If the key presses are not the backspace nor enter key, the *`handle_symbol`* function is executed.

2. **` handle_symbol`** function:

```
handle_symbol:
    mov [bx], al; store the character in the buffer
    inc bx; increment the buffer pointer
    inc cx; increment the character counter
    inc byte [cursor_x]; increment the cursor x coordinate
    pusha; save all registers
    mov ah, 0eh; print the character
    int 10h; Video BIOS interrupt to print the character
    popa; restore all registers
    ret; Return from the function
```

This function displays a character as TTY. The 0eh function of the interruption 10h writes the specified character to the current cursor position and updates the cursor position. When the cursor advances to the end of the last screen line, this scrolls the text up one line. It also expects the character to write in al (which is already there after the int 16h interruption).

The first line, `*mov [bx], al*`, moves the content of the al register into the memory location pointed to by the value in the `bx` register. After that, then pointer is incremented, and so, the counter cx.

Next, the value stored in the memory location indicated by the label `*cursor_x*` is incremented by 1 byte (`*byte*`: this keyword indicates that the memory operation, increment in this case, should be performed on a memory location of size 1 byte; `*[cursor_x]*`: this is a memory reference using the `*cursor_x*` label. It represents the memory location where the x-coordinate of the cursor is stored).

The following line, `*pusha*`, is used to push the values of all the general-purpose registers onto the stack, and be able to change the values of the registers inside the block between it and `*popa*`, when the values will be restored to the ones before `*pusha*`. This is used to be able to use the 0eh function of the 10h interruption.

3. **`handle_backspace`** function:

```
handle_backspace:
    ; If already at the top line, do nothing
    cmp byte [cursor_coords], 0
    je .backspace_done

    ; If character counter is 0, go on previous line
    cmp cx, 0
    je .backspace_prev_line

    dec bx; decrement the buffer pointer
    dec cx; decrement the character counter
    dec byte [cursor_x]; decrement the cursor x coordinate
    pusha; save all registers
    mov ah, 02H; set the cursor position
    mov bh, 0; page number
    mov dx, [cursor_coords]; cursor coordinates
    int 10h

    mov ah, 0AH; print the character at the cursor position
    mov bh, 0; page number
    mov cx, 2; number of times to print the character
    mov al, ' '; print a space
    int 10h
    popa; restore all registers
    ret

    .backspace_prev_line:
    dec byte [row]; decrement the row number
    dec byte [cursor_y]; decrement the cursor y coordinate
    mov ah, 02H; set the cursor position
    mov bh, 0; page number
    mov dx, [cursor_coords]; cursor coordinates
    int 10h
    ret

    .backspace_done:
    cmp byte [row], 0
    jne .backspace_prev_line
    ret
```

The next function, handles the behavior of the backspace key. First of all, it checks if the values of the `*cursor_coords*` are equal to 0, to check if the cursor is on the edge of the screen. If yes, then the function jumps to the `*.backspace_done*`, checks if the cursor is also on the first row, and if yes, than the function simply returns.

If the character counter is zero, but the cursor is not in the first line, then it must go to a previous line, and this functionality is handled by the `*.backspace_prev_line*`. The function decrements the values of the row and the cursor y coordinate, and after that, executes the function 02H of the interruption 10h, setting the cursor position, which expects the page number to be set (0 in my case), and the values of the position of row and column (`*cursor_coords*`)

If the characters counter is not zero, then the functionality of the backspace key is executed: the buffer pointer is decremented, the character counter is also decremented and the cursor x coordinate id also decremented. Next, the values of all registers are saved.

Next, the function 02H of the interruption 10h is executed. This way, the cursor is set to the coordinates `*cursor_coords*`, in which the value of x coordinate is decremented from earlier, thus the cursor is moving one position back. The registers are restored then, and the function returns.

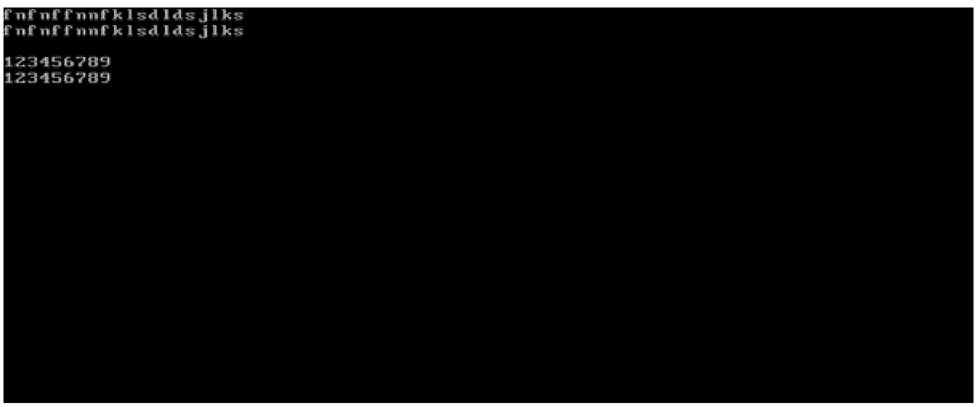
4. **`handle_enter`** function:

```
handle_enter:
    call next_line

    cmp cx, 0
    je .enter_done

    call print_string

    .enter_done:
    ret

next_line:
    inc byte [row]; increment the row number
    inc byte [cursor_y]; increment the cursor y coordinate
    pusha
    mov ah, 02H; set the cursor position
    mov bh, 0; page number
    mov dx, [cursor_coords]; cursor coordinates
    int 10h
    popa
    ret

print_string:
    mov ax, 1300H; print string
    mov bh, 0; page number
    mov bl, 07H; text attribute
    mov dh, [row];
    mov dl, 0; column
    mov bp, buffer; pointer to string
    int 10h

    mov byte [cursor_x], 0; set the cursor x coordinate to 0
    call next_line
    call next_line

    mov bx, buffer; set the buffer pointer to the beginning of the buffer
    mov cx, 0

    ret
```

Next, there is the handling of the enter key. No matter whether the count of characters is zero or not, the enter key should make the cursor go the next line at least once. That’s why, first of all, the function `*next_line*` is called. This function increments the row and y coordinate of cursor values. After that, the function sets the position of the cursor to be on the next line.

If the characters counter is zero, then the function does nothing, and otherwise, it prints the string from the buffer, using the function 13h of the interruption 10h. It expects the page number, attribute, coordinates of row and column, and the address of start of text to write. The function does not change the cursor position. After that, the function `*next_line*` is called twice. Then, the buffer pointer is set to the beginning of the buffer, and the counter is set to 0.

Results:



Conclusion:

Overall, the study of functions and interrupts in this laboratory session formed the foundation for understanding structured assembly programming. This knowledge encompassed the utilization of interrupts to interact with hardware, the creation of modularized code using functions, and the integration of these components to build a functional program capable of keyboard input processing and screen display management.